Programownie I
Wykład 6

dr inż. Adam Zielonka

Instytut Matematyki,
Politechnika Śląska

Gliwice 23.11.2018

Wskaźniki

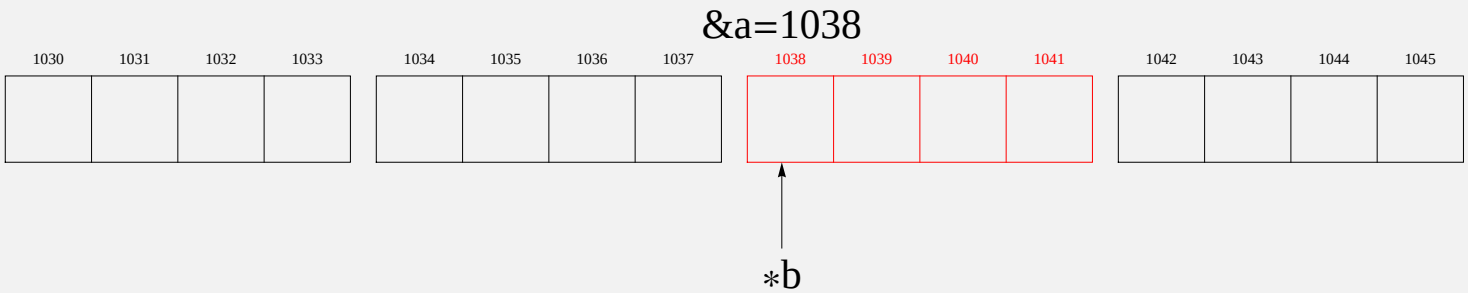
```
int a=453;//zajmuje 4 kolejne bajty pamięci
```

&a adres pierwszej komórki pamięci, w której "przechowywana" jest wartość zmiennej a

Deklaracja wskaźnika

```
int *b = nullptr; //wskaźnik do zmiennej typu int
```

```
b=&a; //b wskazuje na zmienną a
```



Wskaźniki

```
int a=5,b=3,c[10]{};

int *w=nullptr;

w=&a;  //wskazuje na a

b=*w;  //wyłuskanie wartości spod adresu w

*w=7;  //zmiana wartości pod adresem w na 7

w=&c[2]; //ustawienie wskaźnika na c[2]
```

Wskaźniki a tablice

```
int tablica[] = {8, 8, 6, -9, 12};
//wskaznik do tablicy
int *wsk = &tablica[0];

cout << *wsk << endl;

cout << *(wsk+1) << endl;

cout << tablica[2] << endl;
```

`*(wsk+n)` oznacza wyłuskanie elementy typu wskaźnika znajdującego się pod adresem `wsk+n*sizeof(typ)`
W przypadku tablicy zapis ten jest równoważny `tablica[n]`

Wskaźniki a tablice

Nazwa tablicy jest wskaźnikiem, który nie może zmieniać wartości, czyli zapamiętanego w nim adresu. Zatem zamiast:

```
int tab[] = { 10, 2, 3, 4 };
int *wsk = &tab[0];
```

można wywołać:

```
int tab[] = { 10, 2, 3, 4 };
int *wsk = tab;
```

Wskaźniki

```
int tab[] = { 10, 2, 3, 4 };
int *wsk = tab;

wsk++; //ustawienie wskaźnika na adres kolejnej
       //zmiennej typu int, czyli na wyraz tab[1]

//tab++; // błąd, ten wskaźnik nie może ulec zmianie

cout << *(wsk+1); //wypisze 3

cout << *(tab+1); //wypisze 2

cout << *++wsk;
```

Przekazywanie tablicy prostokątnej do funkcji

```
void drukuj( int *tab, int n, int m)
{
    for(int i=0 ; i < n ; i++ )
        for(int j=0;j<m ; j++ )
            cout << *(tab+i*n+j) << ' ';
    cout << endl;
}
```

```
int main()
{
    int[][2] tab={ {2, 4} , {3, 4} , {0, 1} };
    drukuj(&tab[0][0], 3 , 2);
    system("PAUSE");
}
```

Przekazywanie tablicy prostokątnej do funkcji

```
void drukuj( int tab[][2], int n)
{
    for(int i=0 ; i < n ; i++ )
        for(int j=0;j<2 ; j++ )
            cout << tab[i][j] << ' ';
    cout << endl;
}
```

```
int main()
{
    int[][2] tab={ {2, 4} , {3, 4} , {0, 1} };
    drukuj( tab , 3 );
    system("PAUSE");
}
```

Rzutowanie wskaźników

```
unsigned int tabInt[] = { 11,123,9999999 };
unsigned int *wskUI = tabInt;

for (int i = 0; i < 3; i++)
    cout << (*(wskUI + i)) << endl;

unsigned char *wsk_UCh = nullptr;

wsk_UCh = reinterpret_cast<unsigned char*>(wskUI);

auto wsk_Ch = reinterpret_cast<char *>(wskUI);

for (int i = 0; i < 12; i++)
    cout << (*(wsk_UCh + i)) << endl;
```

Wskaźnik typu void*

Wskaźnik typu void* przekazuje tylko adres, nie zawiera informacji o typie danych, na które wskazuje.

```
int tab[10]={1,1,1,1};

int *wsk_int=&tab[9];

void *wsk_v=wsk_int;
...
wsk_int=reinterpret_cast<int*>( wsk_v );
```

Dynamiczne lokowanie pamięci w języku C

Funkcje calloc i malloc

```
void * calloc(ilosc obiektow, rozmiar obiektu);
//inicjuje pamięc zerami


void * malloc(rozmiar_w_bajtach);
```

Przykład

```
int *wsk_int=( int* )calloc ( 1000, sizeof(int) );
...
free(wsk);

void *wsk_v = malloc(1000000);
...
free(wsk_v);
```

Dynamiczne lokowanie pamięci w języku C

Przykład

```
int n=100000;
int *p=( int* )malloc ( n * sizeof(int) );

p[10]=12;

cout << (*(p+10))<<endl;

free(p);
```

Uwaga!!! Pamięć dynamicznie przydzielona nie jest automatycznie zwalniana, nawet jeśli zmienna lokalna p przestanie istnieć. Funkcja free zwalnia pamięć wskazywaną przez p, lub nic nie robi jeśli p jest nullptr. Można zwalniać jedynie bloki pamięci spod adresów uzyskanych funkcjami malloc lub calloc.

Dynamiczne lokowanie pamięci w języku C

```
void funkcja(void)
{
 int *wsk = malloc( 8 * sizeof( int ) );
}
int main()
{
 for(int i = 0 ; i < 1000 ; i++ )
   funkcja();
}
```

Jeśli „zgubimy adres dynamicznie przydzielonej pamięci, to już jej nie zwolnimy.

Dynamiczne lokowanie pamięci w języku C++

new delete

```
int n=500000;
int *wsk = new int;
int *tab = new int[n];
...
delete wsk;
delete [] tab;
```

Przykład

```
int n{};
cout << "Podaj dlugosc wektora n=";
cin >> n;
int * tablica = new int [n];
for( int i=0; i<n; i++ )
{
    cout << "v[" << i <<"]=";
    cin >> tablica[i];
}
...
delete [] tablica;
```

Tablice wskaźników

c-stringi

```
char *dniPL[] = { "poniedzialek" , "wtorek" , "sroda" ,
                  "czwartek", "piatek"};

char *dniEn[]={"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};

int dzien=2;

char **dni{};
int jezyk=0;
    if(jezyk==0)
        dni=dniPL;
    else
        dni=dniEn;

cout << *(dni+dzien)<<endl;
```

Argumenty wejścia

```
nazwa_programu.exe arg1 arg2
```

```
int main(int argc, char* argv[])
{
    ...
}
```

argc - ilość podanych argumentów, równa jeden gdy nie podano argumentów (wlicza nazwę programu)
argv[] - wskaźniki do argumentów, gdzie:
argv[0] - nazwa programu,
argv[1] - argument pierwszy,
...

Argumenty wejścia

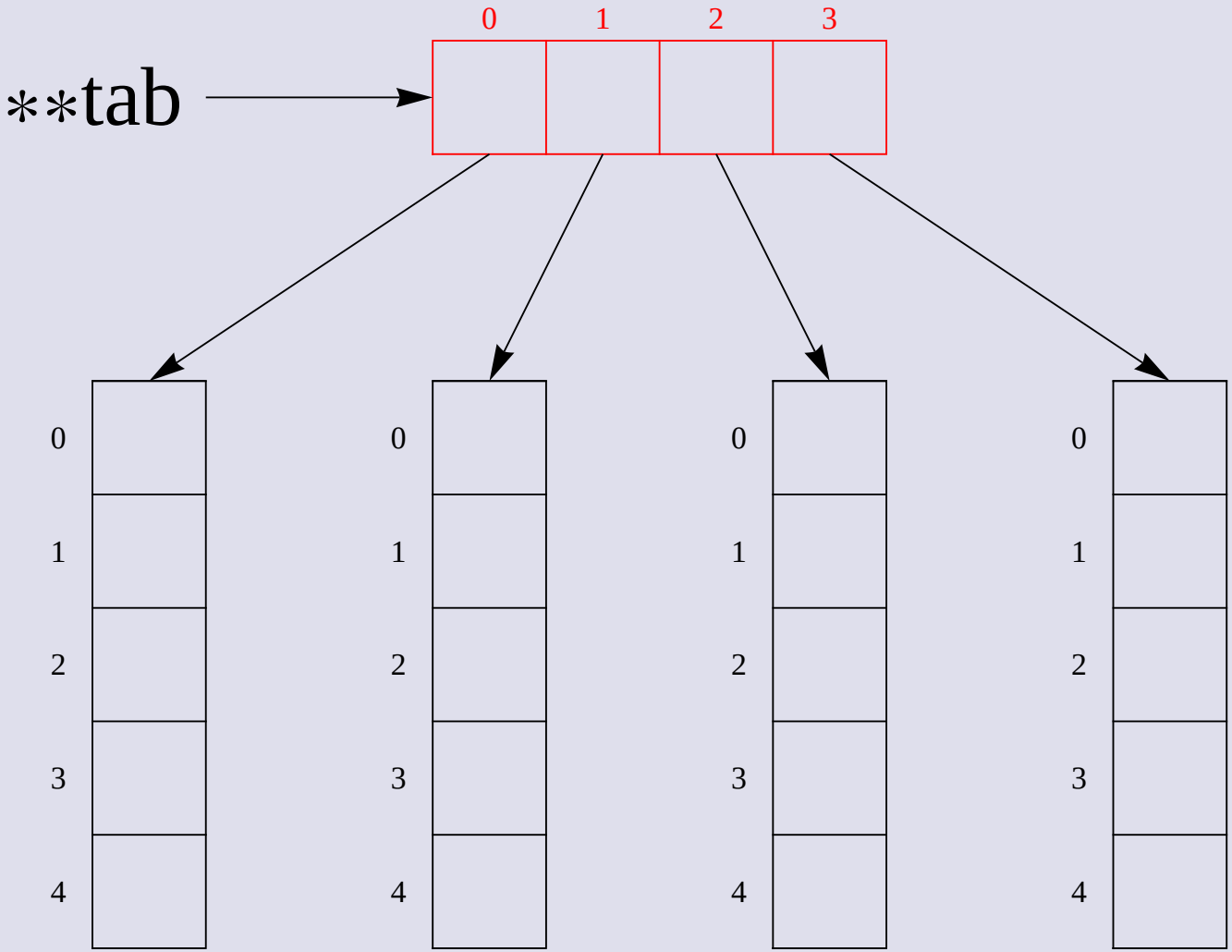
```
#include<iostream>
#include<cstdlib>
int main(int argc, char* argv[])
{
    if(arc<3)
    {
        cout << "Za mała liczba argumentów " << endl;
        system("PAUSE");
        return 1;
    }
    int suma=0;
    for(int i = 1 ; i < argc ; i++)
        suma+=atoi(argv[i]);
    cout << suma << endl;
    system("PAUSE");
    return 0;
}
```

Tablice wskaźników

Tablica dwuwymiarowa

```
int nA=4,nB=6, **A{};
A = new int *[nA];
for (int i = 0; i < nA; i++)
        *(A+i) = new int[nB];
...
//
for (int i = 0; i < nA; i++)
        delete[] A[i];
delete[] A;
```

Tablice wskaźników



Inteligentne wskaźniki

Unikalny wskaźnik unique pointer

```
#include<memory>
...
unique_ptr<T[]>nazwa(new T[n]);
```

- Jest jedynym odniesieniem do tej pamięci, na którą wskazuje.
- Wartość unikalnego wskaźnika nie może być kopiowana.
- Pamięć zarezerwowana przez niego zostanie zwolniona w chwili jego zniszczenia.

```
int n=9000;
unique_ptr<int[]> tab(new int[n]);
```

Inteligentne wskaźniki

Wskaźnik współdzielony shared pointer

```
#include<memory>
...
shared_ptr<T>nazwa1(new T);
shared_ptr<T>nazwa2;
...
nazwa2=nazwa1; //poprawne
```

- W przeciwieństwie do `unique_ptr<>` wskaźniki współdzielone można kopiować.
- Wskaźniki 'wiedzą' o współistniejących kopiach.
- Pamięć zarezerwowana przez nie zostanie zwolniona w chwili usunięcia ostatniego z nich.

Wskaźnik do funkcji

Deklaracja i definicja

```
int funkcja(void);
int funkcja2(int, int);
...
//wskaźnik do funkcji o prototypie void f(void)
int (*wFun1)()=nullptr;
wFun1=&funkcja;

//wskaźnik do funkcji o prototypie int f(int, int)
int (*wskFun2)(int, int){};
wskFun2=&funkcja2;
```

Wskaźnik do funkcji

Deklaracja i definicja

```
int funkcja(void);
int funkcja2(int, int);
...
//wskaźnik do funkcji o prototypie void f(void)
int (*wFun1)()=nullptr;
wFun1=&funkcja;

//wskaźnik do funkcji o prototypie int f(int, int)
int (*wskFun2)(int, int){};
wskFun2=&funkcja2;
```

Wskaźnik do funkcji

Przykład

```
int suma(int, int);
int iloczyn(int, int);
int main()
{
    int(*dzialanie)(int, int) {};

    int operacja = 1;
    int a = 10, b = 21;
    if (operacja == 1)
        dzialanie = &suma;
    else
        dzialanie=&iloczyn;
    cout << dzialanie(a, b);
}
int suma(int a, int b){return a+b;}
int iloczyn(int a, int b){return a*b;}
```

Wskaźnik do funkcji jako argument funkcji

Przykład

```
double normaL2(double *, int);
double normaL1(double *, int);
double normaM(double( * )(double*, int), double**, int, int);
int main()
{
    double** mac;
mac = new double*[2];
mac[0] = new double[2]{ 1,2 };
mac[1] = new double[2]{ 3,2 };
cout << endl << "L1=";
cout<<normaM(normaL1,mac , 2, 2);
cout << endl << "L2=";
cout << normaM(normaL2, mac, 2, 2);
}
double normaM(double(*normaW)(double*, int), double** mac, int n, int m)
{
    double* wek = new double[n] {};
    for (int i = 0; i < n; i++)
        *(wek+i) = normaW(*(mac + i), m);
    return normaW(wek,n);
}
```